N90-27322

Construction of Dynamic Stochastic Simulation Models Using Knowledge-Based Techniques

M. Douglas Williams
Advanced Technology, Inc.
555 Sparkman Dr., Suite 454
Huntsville, AL 35816

Sajjan G. Shiva
Computer Science Department
University of Alabama in Huntsville
Huntsville, AL 35899

ABSTRACT

Over the past three decades, computer-based simulation models have proven themselves to be cost-effective alternatives to the more structured deterministic methods of systems analysis. During this time, many techniques, tools and languages for constructing computer-based simulation models have been developed. More recently, advanced in knowledge-based system technology have led many researchers to note the similarities between knowledge-based programming and simulation technologies and to investigate the potential application of knowledge-based programming techniques to simulation modeling.

This paper discusses the integration of conventional simulation techniques with knowledge-based programming techniques to provide a development environment for constructing knowledge-based simulation models. A comparison of the techniques used in the construction of dynamic stochastic simulation models and those used in the construction of knowledge-based systems provides the requirements for the environment. This leads to the design and implementation of a knowledge-based simulation development development environment.

These techniques have been used in the construction of several knowledge-based simulation models including the Advanced Launch System Model (ALSYM).

1.0 INTRODUCTION

Knowledge-based simulation extends the set of tools available to the simulation modeler by incorporating techniques from the field of artificial intelligence. [Smith et al 1988] In the context of this research, simulation techniques are limited to discrete-event simulation methods for constructing dynamic stochastic simulation models. The techniques from

artificial intelligence that have proved most useful are broadly classified as knowledge-based programming techniques, hence the name knowledge-based simulation. There are many similarities between knowledge-based programming techniques and conventional discrete-event simulation techniques, as well as some important differences.

The most important similarity is the separation of the domain problem-solving knowledge from the control strategy applying this knowledge to solve some problem instance in the domain. In knowledge-based programming, this knowledge is represented in the form of rules or logical statements. The control strategy is implemented by an inference engine which repeatedly chooses the most appropriate of these rules or logical statements based on the current state of the system. The control strategy then performs the actions specified by that rule or logical statement to effect a change in the state of the system. In discrete-event simulation models, the knowledge is represented in the form of events and the control strategy repeatedly chooses the next imminent event, i.e. the one which is to occur next in simulated time, advances the simulation clock to the time the imminent event is scheduled to occur, and executes the code associated with the event to effect a change in the state of the system. In both cases, this allows the programmer (i.e. knowledge engineer or simulation modeler) to concentrate on developing the domain knowledge without having to worry about the exact order in which the individual chunks of knowledge are applied in solving a particular problem instance.

Another important similarity is that both fields use similar methodologies for representing the current state of the system. This is most often some variation of the entity-attribute approach. The entity-attribute approach was developed as a modeling technique in which the system is decomposed into its constituent components which are called entities. Each entity is then modeled in terms of its attributes where each attribute represents some aspect of the entity. In knowledge-based systems, the entity-attribute approach serves as the basis for the frames knowledge representation technique which extends the entity-attribute approach by modeling attributes in terms of their facets. Each facet of an attribute represents a different aspect of the attribute, and may include precedural attachments called daemons which are activated in response to references to the attribute. In addition, operations on an entity are specified as procedural attachments on the entity itself and are activated in response to messages directed at the entity. A programming system based on frames is called an object-oriented programming system.

Finally, an iterative development methodology is regarded by both fields as the appropriate approach to system development. In discussing his methodology of simulation model development, Shannon stated that "the evolutionary nature of model building is inevitable and desirable" [Shannon 1975]. This is comparable to the accepted knowledge engineering methodology where the "system evolves by proceeding from simple to increasingly hard tasks, improving incrementally the organization and representation of knowledge" [Hayes-Roth 1983]. The iterative development

methodology has been extended by knowledge-based development environments to provide rapid prototyping of knowledge-based systems.

Figure 1 summarizes the similarities between conventional discrete-event simulation techniques and knowledge-based programming techniques.

One important difference between knowledge-based programming techniques and conventional discrete-event simulation techniques is the development environments within which each is typically implemented. Knowledge-based programming tools and languages are typically implemented using, or as extensions to, symbolic programming languages with interactive development environments on personal workstations, whereas conventional discrete-event simulation tools and languages are typically implemented using, or as extensions to, conventional programming languages using modest development environments provided by traditional mini-computer/main-frame operating systems. The fact that these conventional programming languages can be efficiently translated into executable machine code while the symbolic programming languages rely on either an interpretive execution environment or a non-conventional hardware architechure has hindered the integration of knowledge-based techniques into conventional programming environments, and vice versa.

The interactive development environments within which knowledge-based languages and tools exist rely on the interpretive nature of symbolic programming languages to allow changes in the program to be immediately reflected in the system being developed. This allows knowledge-based systems to support the iterative development method, not only at a macro level, but at all levels of development. This is the basis of the popularity of the use of rapid prototyping techniques in the development of knowledge-based systems.

Conventional Discrete-Event Simulation Techniques	Knowledge-Based Programming Techniques
• Domain knowledge separate from control strategy	• Domain knowledge separate from control strategy
• Entity-attribute data organization	• Object-oriented data organization with active objects
• Iterative development	• Iterative development with rapid prototyping

Figure 1. Similarities Between Conventional Discrete-Event Simulation and Knowledge-Based Programming Techniques

Another important difference between knowledge-based programming techniques and conventional discrete-event simulation techniques is their ability to deal with the concept of time within their respective domains. The concept of time is an inherent part of dynamic simulation models. The discrete-event simulation techniques that have been developed to construct such models are centered around providing an appropriate abstraction of time. On the other hand, the inferencing mechanisms used within knowledge-based systems, e.g. rule-based or logic-based approaches, are typically confounded by time-varying data. Dealing with these problems requires the specification of additional meta-knowledge which defines the dynamic effects of changes to time-varying data used in deducing other data.

Figure 2 summarizes the differences between conventional discrete-event simulation techniques and knowledge-based programming techniques.

Conventional discrete-event simulation techniques and knowledge-based programming techniques complement each other when integrated into a unified knowledge-based simulation development environment. The knowledge-based programming techniques provide the enhanced data abstraction capabilities associated with object-oriented programming techniques, rule-based or logic-based inferencing capabilities for dealing with complex decision processes, and a superior development environment for supporting incremental development. The conventional discrete-event simulation techniques provide the basic entity-attribute approach for data abstraction and the capabilities to deal with time-varying data within the domain.

2.0 Design of a Knowledge-Based Simulation Environment

There are many techniques from conventional simulation modeling and knowledge-based programming which are applicable to knowledge-based simulation modeling. Applicable conventional simulation modeling techniques include:

- the entity-attribute approach,
- the event-scheduling control strategy, and
- the process-interaction control strategy.

Conventional Discrete-Event Simulation Techniques	Knowledge-Based Simulation Techniques
• Event scheduling approach to control	• Rule-based or logic-based inferencing
• Time central concept in control strategy	• Time normally not used in inferencing
• Quantitative models (i.e., primarily numeric data)	• Qualitative models (i.e., primarily symbolic data)
• Model data probabilistic	• Model data have associated certainty factors
• Model results probabilistic, model is run many time and results distributions computed	• Model results deterministic, may have associated certainty factors

Figure 2. Differences Between Conventional Discrete-Event Simulation and Knowledge-Based Programming Techniques

Applicable knowledge-based program-ming techniques include:

- object-oriented programming,
- rule-based programming,
- forward-chaining inferencing, and
- backward-chaining inferencing.

These techniques must be analyzed to eliminate redundancy and to insure compatability when integrated into a knowledge-based simulation development environment.

The entity-attribute approach and object-oriented programming both provide for the representation of data in their respective domains. Object-oriented programming provides all of the techniques needed to represent model entities, their attributes, and operations on them in a unified manner. Object-oriented programming therefore provides all of the capabilities of the entity-attribute approach and extends it to include procedural operations on entities.

Knowledge-based simulation models must be able to deal adequately with the concept of time. This requires the inclusion of a simulation clock and a mechanism for maintaining its value. The event-scheduling control

strategy is the obvious choice for providing these timing functions. An alternative approach would be to put the simulation clock on the blackboard and allow rules to access and/or update its value. This would allow rule-based programming to be used for maintaining the value of the simulation clock. This approach, however, lacks the clarity and efficiency of the event-scheduling approach.

The process-interaction approach unifies the event-scheduling control strategy and object-oriented programming. A process represents an active entity in the model and is represented in the same manner as any other entity using object-oriented programming. In addition to having attributes and operations specified, a process has associated events which define its dynamic behavior. Individual events within a process may communicate via the attributes of its associated process instance.

Rule-based programming is used to model complex decision-making processes within a knowledge-based simulation model. All decisions are assumed to be instantaneous in simulated time, therefore rules must not use the simulation clock in any of their preconditions and must not alter the value of the simulation clock in any of their actions. This precludes the possibility of any conflicts between the event-scheduling control strategy and the inferencing control strategy. Any event may initiate an inferencing procedure to model a decision-making process, and any rule action may schedule events to be executed or modify the future event list to effect changes in future event execution.

Either forward-chaining or backward-chaining, or both, may be used in implementing an inferencing control strategy. It is possible to structure rules such that the same rule may be used in both forward-chaining and backward-chaining inferencing. However, because backward-chaining requires the ability to identify the facts that a rule's actions may place on the blackboard, this can only be done by restricting the actions that a rule is allowed to perform. The prototype development environment is therefore restricted to using a forward-chaining inferencing mechanism in its inference engine.

The language for the knowledge-based simulation model development environment is based on the Common Lisp programming language. There are several reasons for this choice:

1. Common Lisp provides features to support the embedding of new language features within the language, most notably, the macro facility which allows new special forms to be added to the language.

2. There are object-oriented programming systems available within Common Lisp implementations. One of these can serve as the basis for the object-oriented programming system.

3. There are sophisticated development environments available which support incremental development and rapid prototyping.

The Common Lisp implementation used is Symbolics Common Lisp which runs on the Symbolics Lisp Machine.

The language elements that comprise the knowledge-based simulation development environment are divided into five protocols:

1. The modeling protocol provides the language elements for the object-oriented programming system for the representation of model elements.

2. The simulation protocol provides the language elements to extend the modeling protocol for the definition of the dynamic simulation elements.

3. The simulation control protocol provides the language elements for the control of the dynamic simulation elements.

4. The inferencing protocol provides the language elements for the rule-based programming techniques for the modeling of complex decision processes.

5. The inferencing control protocol provides the language elements for the control of the forward-chaining inferencing mechanism.

Each of these protocols will be discussed in the following sections.

2.1 Modeling Protocol

The modeling protocol provides the static modeling language elements. The modeling protocol is provided by the Flavors package which is an object-oriented programming package implemented on top of Common Lisp. A complete description can be found in the Symbolics Common Lisp Language Concepts Vol. 2A. [Bromley et al 1987; Symbolics 1988a; Symbolics 1988b]

The object-oriented programming package provided by the Flavors package uses terminology that differs from similar packages. Entities classes are represented by flavors. A flavor serves as a template for all objects in the corresponding entity class. Each object is an instance of a flavor. The entity attributes for a flavor are represented by instance variables. Each instance of a flavor has its own values for its instance variables, and the values of these instance variables define the state of the instance. Each instance variable may have a default initial value and may be defined to be inittable, i.e., its value may be specified when an instance is created; readable, i.e., its value may be read via a function call; and/or writable, i.e., its value may be updated via a call to the `setf` macro. There are no class variables for a flavor, though the same concept may be easily

emulated using the property list of the symbol naming the flavor. [Bromley et al 1987; Symbolics 1988a; Symbolics 1988b]

The operations for the entity class represented by a flavor are defined as methods on the flavor. A method is a function associated with a flavor. When the method is activated for a specific flavor instance, the variable `self` is bound to the instance and all of the instance variables for the instance are available to the method as local variables. [Bromley et al 1987; Symbolics 1988a; Symbolics 1988b]

One of the main strengths of the Flavors package is the ability to combine existing flavors with newly defined flavors. When defining a flavor, a list of component flavors whose characteristics are to be included in the new flavor is specified. The instance variables and methods of the component flavors are inherited by the new flavor. [Bromley et al 1987; Symbolics 1988a; Symbolics 1988b]

2.2 Simulation Protocol

The simulation protocol provides the dynamic modeling language elements. These language elements provide for the definition of processes and the events which implement their actions. The protocol also provides language elements for process creation, interprocess communication, and other operations on processes and events. The simulation protocol implements a process interaction approach to discrete event simulation.

A process represents an active entity in the reference system. Processes may have instance variables to represent attributes unique to each process instance like other entities. A process is defined using the `defprocess` macro. A call to this macro has the following form:

```
defprocess name
    &rest instance-variables
```

where *name* is a symbol that is the name of the process being defined, and each *instance-variable* is the name of an attribute of the process and has the same format as an instance variable specification in a flavor definition. The primary action of the macro is to define a flavor whose name is *name* based on the abstract flavor `process`. The macro also defines a predicate to recognize instances of the process. The name of this predicate function is the print name of *name* with "-P" appended.

The simulation protocol has two mechanisms for creating process instances. The more primitive mechanism for creating a process instance is the `create-process` function. This function creates a new process instance and optionally initializes some instance variables in the newly created process instance. A call to this function has the following form:

```
create-process name
    &rest initializations
```

where *name* is the name of the process of which to create an instance, and the *initializations* are optional keyword/value pairs specifying initial values for process instance variables. The newly created process instance is returned as the value of the function call.

Most processes have an initial event which is scheduled to execute as the first event after the process is created. The simulation protocol provides an `initiate` macro which simplifies the normal process of process creation and initial event scheduling. The `initiate` macro creates a new process instance, optionally initializes some instance variables in the newly created process instance, and schedules its initial event for execution at some specified simulated time. A call to this macro has the following form:

```
initiate time name
    &rest initializations
```

where *time* is the simulated time at which the initial event of the process is to be executed or `:now` if the initial event is to be scheduled to execute immediately, *name* is the name of the process to be initiated, and the *initializations* are optional keyword/value pairs specifying initial values for process instance variables. Note that the *name* argument is not evaluated. The newly created process instance is returned as the value of the macro call.

The actions of an active entity are specified as events within the process representing the active entity. Each event has a parameter list and accepts argument like a normal Common Lisp function, but rather than being executed immediately in response to a function call their executions are scheduled and performed in their proper sequence by the event-scheduling control strategy. Note that a process may also have normal methods defined on them using the `defmethod` macro. An event is defined using the `defevent` macro. There are two types of events: process events and non-process events. As the names suggest, a process event is an event which is associated with some process, whereas a non-process event is not. A call to this macro has the following form:

```
defevent
    (name
      &optional process
      &key :initial-event)
    lambda-list
  &body body
```

where *name* is a symbol that is the name of the event being defined, *process* is the name of the process with which this event is associated or `nil` (the default) for a non-process event, the key `:initial-event` is true if this event is the event to be scheduled when the associated *process* is initiated and `nil` (the default) otherwise. The *lambda-list* specifies the arguments to this event and may contain any structure valid in a function

lambda list. The *body* specifies the forms that are to be executed as the body of this event.

There are often cases where an event executing within a process instance is required to communicate with another process instance. This may be to read or update an instance variable within the referenced process instance or to schedule an event within the referenced process instance. A common example of this occurs when there is some initial handshaking operations that must be performed between a process instance and its initiator. The `with-process` macro provides a convenient mechanism for performing these actions. A call to this macro has the following form:

```
with-process
  (process
    &optional instance-form)
  &body body
```

where *process* is the symbol naming the process whose instance is required, and *instance-form* is an optional form which, when evaluated, returns the desired process instance. The *body* specifies the forms that are to be evaluated with *process* bound to the referenced process instance. If an *instance-form* is not given, or is `nil`, then *process* is bound to the ancestor process of the indicated type.

The final language element provided by the simulation protocol is a macro to iterate over the events defined within a process. The do-process-events macro provides this iteration capability. A call to this macro has the following form:

```
do-process-events
    (var process
        &optional result-form)
  &body body
```

The effect of a call to the `do-process-events` macro is to evaluate the forms in *body* with the symbol *var* bound to successive events in *process*. After all of the events have been exhausted, *result-form*, which defaults to `nil`, is evaluated and the result returned as the value of the call.

2.3 Simulation Control Protocol

The simulation control protocol provides the language elements for the creation and manipulation of simulation environments. The most important elements of the simulation environment are the future event list and the simulation clock. A simulation environment provides the data structures for the process interaction control strategy of a simulation model. Multiple simulation environments may exist simultaneously either in a hierarchical (i.e., nested) manner or as independent environments within separate Common Lisp dynamic referencing environments.

A simulation environment automatically exists within each independent Common Lisp dynamic referencing environment. To create a new simulation environment within an existing simulation environment, the `with-new-simulation-environment` macro is used. A call to this macro has the following form:

```
with-new-simulation-environment
    &body body
```

where *body* contains the forms needed to implement the encapsulated simulation model. These forms would normally include one or more process initiations followed by a `start-simulation` function call (see below).

A simulation environment can be reset to its initial state using the `reset-simulation-environment` function. A call to this function has the following form:

```
reset-simulation-environment
```

The effect of a `reset-simulation-environment` function call is to clear the future event list by releasing all of the previously scheduled events and to reset the simulation clock to zero. The effect of the call is limited to the innermost simulation environment in the current Common Lisp dynamic referencing environment.

The process interaction control strategy is initiated within a simulation environment using the `start-simulation` function. A call to this function has the following form:

```
start-simulation
```

The effect of a `start-simulation` function call is to begin the event scheduling control strategy of removing event notices from the future event list, advancing the simulation clock, and executing the corresponding event code. This process continues until either the future event list is empty or a dynamic throw is executed whose catch is not within the dynamic environment of the control strategy. The effect of the call is limited to the innermost simulation environment in the current Common Lisp dynamic referencing environment.

The `schedule` function calls the scheduler to add a new event notice to the future event list representing the future execution of an event. A call to this function has the following form:

```
schedule
  time event process-instance
  &rest arguments
```


C16

where *time* is a number representing the time the event is to be scheduled or :now if the event is to placed at the head of the future event list to be executed at the current simulation time as the next event, *event* is the name of the event to be scheduled, *process-instance* is the process instance which the scheduled event is to be associated, and *arguments* are the actual arguments to the scheduled event. The effect of the call is that an event notice is created representing the future event, and this event notice is placed at the correct position on the future event list.

It is sometimes necessary within a simulation model to make decisions based on already scheduled events, or to remove or reschedule such events. The do-scheduled-events macro provides a mechanism for doing this. A call to this macro has the following form:

```
do-scheduled-events
    (var &optional
         event-list result-form)
  &body body
```

The effect of a call to the do-scheduled-events macro is to evaluate the forms in *body* with the symbol *var* bound to successive event notices on *event-list*, which defaults to the future event list in the current simulation environment. After all of the event notices have been exhausted, *result-form*, which defaults to nil, is evaluated and the result returned as the value of the call.

2.4 Inferencing Protocol

The inferencing protocol provides the language elements for the rule-based modeling of complex decision-making processes. These language elements provide for the definition of rulesets and the rules specifying their problem-solving knowledge. The protocol also provides language elements for ruleset creation and other operations on rulesets ans rules. The inferencing protocol implements a rule-based programming system.

The basic elements of the inferencing protocol are rules and rulesets. A rule is a single piece of problem-solving knowledge expressed as a set of preconditions for the application of the rule and a set of actions that are to be performed when the rule is executed. Related rules are grouped together to form rulesets which may be activated as needed in response to problem-solving demands.

A ruleset represents a body of problem-solving knowledge. A ruleset may also have instance variables to represent attributes unique to each ruleset instance. A ruleset is defined using the defruleset macro. A call to this macro has the following form:

```
defruleset name
    &rest instance-variables
```

where *name* is a symbol that is the name of the ruleset being defined, and each *instance-variable* is the name of an attribute of the ruleset and has the same format as an instance variable specification in a flavor definition. The primary action of a call to the macro is to define a flavor whose name is *name*, based on the abstract flavor `ruleset`. The macro also defines a predicate to recognize instances of the ruleset. The name of this predicate function is the print name of the *name* with "-P" appended.

A ruleset instance must be created to perform inferencing using the rules in the ruleset. The `activate` macro creates a new ruleset instance optionally initializes some instance variables in the newly created ruleset instance, and makes the rules in the ruleset eligible for firing by adding the newly created ruleset instance to the list of active rulesets. A call to this macro has the following form:

```
activate name
    &rest initializations
```

where *name* is the name of the ruleset to be activated, and the *initializations* are optional keyword/value pairs specifying initial values for ruleset instance variables. Note that the *name* argument in an `activate` macro call is not evaluated. The newly created ruleset instance is returned as the value of the macro call.

A rule represents a single piece of problem-solving knowledge and is associated with a particular ruleset. The specification of a rule includes a set of precondition patterns that are to be matched against facts on the blackboard and a set of actions which are forms to be evaluated in an environment binding the variables in the precondition patterns to their matching elements in the matched facts. A rule is defined using the `defrule` macro. A call to this macro has the following form:

```
defrule (name ruleset)
  &rest preconditions-and-actions
```

where *name* is the name of the rule being defined, *ruleset* is the name of the ruleset with which this rule is associated, and the *preconditions-and-actions* have the following form:

```
  {preconditions}*
==>
  {actions}*
```

where each *precondition* is a pattern to be matched against facts on the blackboard, and the *actions* are the forms to be evaluated in an environment binding the variables in the precondition patterns to their matching elements in the match facts when the rule is executed.

The final language element provided by the inferencing protocol is a macro to iterate over the rules defined within a ruleset. The `do-ruleset-rules` macro provides this iteration capability. A call to this macro has the following form:

```
do-ruleset-rules
    (var ruleset
         &optional result-form)
  &body body
```

The effect of a call to the `do-ruleset-rules` macro is to evaluate the forms in *body* with the symbol *var* bound to successive rules in *ruleset*. Note that the *ruleset* argument is evaluated. This is to allow dynamic specification of the ruleset name. After all of the rules have been exhausted, *result-form* which defaults to `nil` is evaluated and the result returned as the value of the call.

2.5 Inferencing Control Protocol

The inferencing control protocol provides the language elements for the creation and manipulation of inferencing environments. The most important elements of the inferencing environment are the blackboard and the agenda. An inferencing environment provides the data structures for the forward-chaining inferencing control strategy for the rule-based components of a knowledge-based simulation model. Multiple inferencing environments may exist simultaneously either in a hierarchical (i.e., nested) manner or as independent environments within separate Common Lisp dynamic referencing environments.

An inferencing environment automatically exists within each independent Common Lisp dynamic referencing environment. To create a new inferencing environment within an existing inferencing environment, the `with-new-inferencing-environment` macro is used. A call to this macro has the following form:

```
with-new-inferencing-environment
    &body body
```

where *body* contains the forms needed to implement the encapsulated rule-based inference system. These forms would normally include one or more ruleset activations and one or more assertions followed by a `start-inferencing` function call (see below).

An inferencing environment can be reset to its initial state using the `reset-inferencing-environment` function. A call to this function has the following form:

```
reset-inferencing-environment
```

The effect of a `reset-inferencing-environment` function call is to clear the blackboard by removing all of the previously asserted facts and to clear the agenda by removing all of the previously triggered rule instances. The effect of the call is limited to the innermost inferencing environment in the current Common Lisp dynamic referencing environment.

The forward-chaining inferencing control strategy is initiated within an inferencing environment using the `start-inferencing` function. A call to this function has the following form:

`start-inferencing`

The effect of a `start-inferencing` function call is to begin the forward-chaining control strategy of matching asserted facts against preconditions of the rules in the active rulesets to form the conflict set, choosing the most appropriate of these rule instances for application, and applying the rule in the appropriate ruleset instance. This process continues until either there are no rule instances in the conflict set after the matching phase of the forward-chaining inferencing algorithm has been executed or a dynamic throw is executed whose catch is not within the dynamic environment of the control strategy. The effect of the call is limited to the innermost inferencing environment in the current Common Lisp dynamic referencing environment.

A fact is a statement about the domain that has either been explicitly given by the knowledge-engineer in the code of the program or by the user in response to some action by the program or has been deduced from these basic facts using the rules in the knowledge base. A fact is represented by a list structure whose structure is determined by the knowledge engineer. The blackboard is a data structure which contains the facts that have been asserted in the current inferencing environment. The `add-fact` function adds a new fact to the blackboard. A call to this function has the following form:

`add-fact` *`fact`*

where *`fact`* is the list structure represention of the fact to be added to the blackboard.

Within the actions of a rule, it is often necessary to remove the fact which matched a particular precondition pattern. The `remove-matching-fact` function removes from the blackboard the assertion representing the fact that matched a given precondition pattern. A call to this function has the following form:

`remove-matching-fact` *n*

where *n* is the number of the precondition pattern whose matching fact is to be removed from the blackboard.

It is sometimes necessary to remove a number of related facts from the blackboard. The `remove-assertions` function removes existing facts from the blackboard that match a given pattern. A call to the `remove-assertions` function removes some facts from the blackboard. A call to this function has the following form:

`remove-assertions` *`pattern`*

where *`pattern`* is a list structure representation of a pattern to match against facts on the blackboard. All facts on the blackboard matching *`pattern`* are removed.

3.0 CONCLUSION

The knowledge-based simulation development language presented has been used in the construction of several knowledge-based simulation models. These include:

- The Advanced Launch System Model (ALSYM) which includes an end-to-end model of the entire Advanced Launch System (ALS) industrial infrastructure.
- The Space Station Freedom Model which provides system available measures for the operational phase of the space station and its support infrastructure.
- The Software Development Process Model which models the performance of a software development organization.

These techniques have provided a capability to develop early prototype models in support of trade studies during the concept development phase of projects as well as detailed models to provide decision support for managers of operational systems.

REFERENCES

Banks, J. and Carson, J. S., II, **Discrete-Event System Simulation**, Prentice-Hall, 1984

Bromley, H. and Lamson, R., **LISP Lore: A Guide to Programming the LISP Machine**, Second Edition, Kluwer Academic Publishers, 1987

Brownston, L., Farrell, R., Kant, E., and Martin, N., **Programming Expert Systems in OPS5**, Addison-Wesley, 1985

Ferguson, E.,*"Using Scheme for Discrete Simulation"*, **Texas Instruments Engineering Journal**, Vol. 3, No. 1, Jan.-Feb. 1986

Hayes-Roth, F., Waterman, D. A., and Lenat, D. B., **Building Expert Systems**, Addison-Wesley, 1983.

Kerckhoffs, E. J. H. and Vansteenkiste, G. C., *"The Impact of Advanced Information Processing on Simulation - An Illustrative Review"*, **Simulation**, Jan. 1986, Vol 46, No. 1, pp. 17-26

Kulikowski, C. A., *"Artificial Intelligence, Modeling, and Simulation"*, **Artificial Intelligence, Expert Systems and Languages in Modeling and Simulation**, North-Holland, Jun. 1987, pp. 5-13

Murray, K. J. and Sheppard, S. V., *"Knowledge-Based Simulation Model Specification"*, **Simulation**, Mar. 1988, Vol. 50, No. 3, pp. 112-119

O'Keefe, R. M., *"The Role of Artificial Intelligence in Discrete-Event Simulation"*, **Artificial Intelligence, Simulation, and Modeling**, John Wiley and Sons, 1989, pp. 359-380

Round, A., *"Knowledge-Based Simulation"*, **The Handbook of Artificial Intelligence**, Vol. IV, Addison-Wesley, 1989, pp. 417 - 518

Shannon, R. E., **System Simulation: The Art and Science**, Prentice Hall, New Jersey, 1975

Shannon, R. E., Mayer, R., and Adelsberger, H. H., *"Expert Systems and Simulation"*, **Simulation**, Jun. 1985, Vol. 44, No. 6

Smith, H.R. and McVicar, K., *"Knowledge-Based Simulation with Frameworks"*, **Proc. of Multiconference 1988**, Feb. 1988

Steele, G. L., Jr., **Common LISP: The Language**, Digital Press, 1984

Symbolics Common LISP - Language Concepts, Symbolics, Inc., 1988

Symbolics Common LISP - Language Dictionary, Symbolics, Inc., 1988

Waterman, D. A., **A Guide to Expert Systems**, Addison-Wesley, 1985

Widman, L. E. and Loparo, K. A., *"Artificial Intelligence, Simulation, and Modeling: A Critical Survey"*, **Artificial Intelligence, Simulation, and Modeling**, John Wiley and Sons, 1989, pp. 1-44

Williams, M. D. and Smith, M. E., *"Knowledge-Based Simulation"*, **Proc. of the Second International Software for Strategic Systems Conference**, Oct. 1988